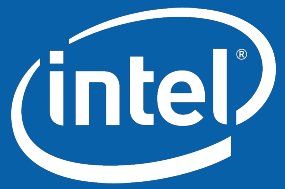


Intel® Itanium® Architecture Basics

Cameron McNairy
Itanium® Processor Architect
Intel® Corporation

Objectives

Understand the principles of EPIC

Introduce the features of the Intel® Itanium® Architecture

Introduce software pipelining

Introduce the Montecito processor

EPIC Architecture Principles

EPIC: Explicitly Parallel Instruction Computing

"The *compiler* should play the key role in designing the plan of execution, and the architecture should provide the requisite support for it to do so successfully;

The Architecture should provide features that assist the *compiler* in exploiting statistical ILP; And

The Architecture should provide mechanisms to communicate the *compiler's* plan of execution to the hardware."

Schlansker, Michael S. and Rau, B. Ramakrishna (HP Labs),
"EPIC: Explicitly Parallel Instruction Computing"
Computer, Vol 33 Issue: 2 , Feb. 2000 pp 37-45

Building Blocks and Background

Question: How to address the needs of the high end server market?

- Large scalability, high performance, adaptability, high availability and resiliency

Answer: Itanium® Processor Architecture

- Application architecture emphasizes program performance and scalability
- System architecture emphasizes adaptability, scalability, availability and resiliency

- Application Architecture – great performance with opportunities to do better
  - Code generator identifies and forms parallelism (EPIC)
  - Simplified in order execution
  - Allow out of order memory hierarchy
- System Architecture – adaptable to multiple OSes, OEMs, and customers
  - Flexible, capable, large memory management and accessibility
  - Bi-endian, OS specific registers, efficient OS calling mechanisms
  - RAS+M for enterprise and mainframe with Machine Check Architecture

Key Architectural Features

Instruction-level parallelism

Large number of registers

Register stack

Predication

Speculation

Floating Point Architecture

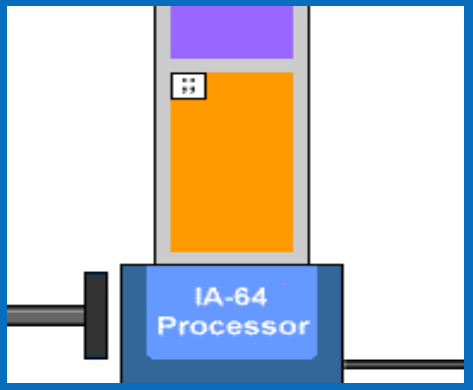
Memory Access

Control Flow

Software pipelining

Intel® Itanium® 2 Processor

Instruction Level Parallelism

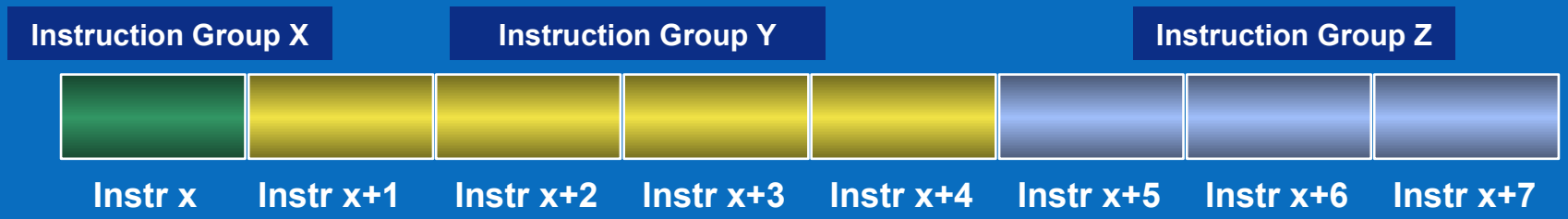


Instruction Groups

An instruction group is one or more independent instructions that may be executed concurrently

Read after write (RAW) or write after write (WAW) dependencies block concurrence

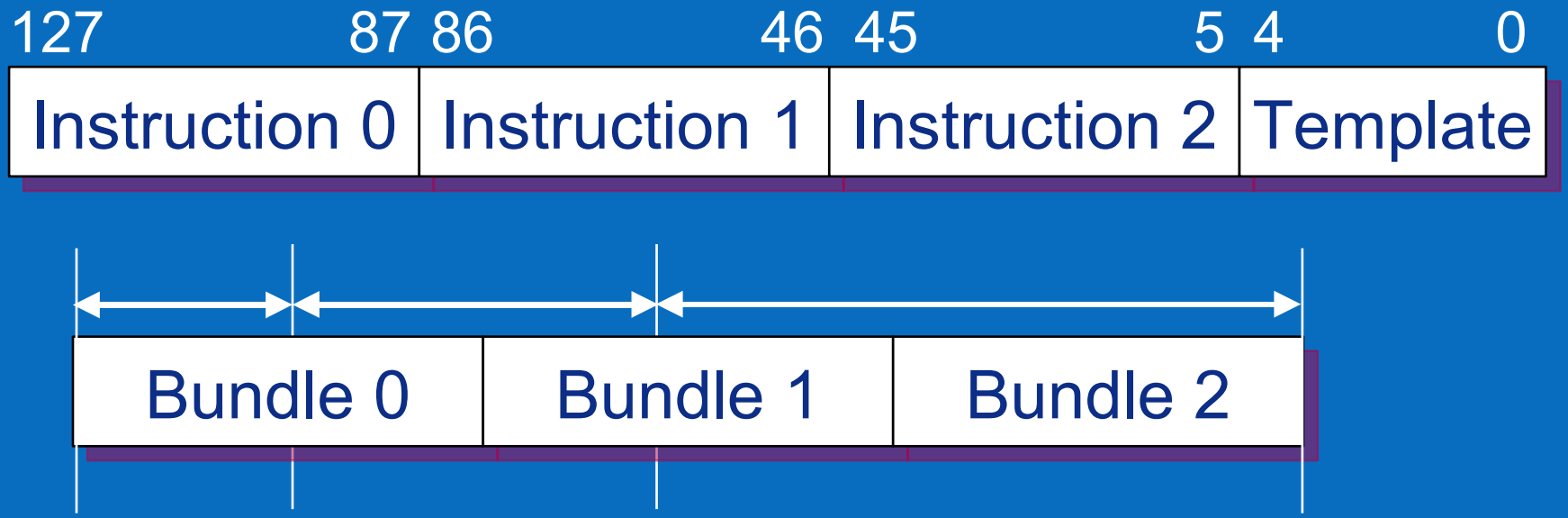
Instruction groups issue in parallel, depending on available resources



Instruction Level Parallelism

Bundles

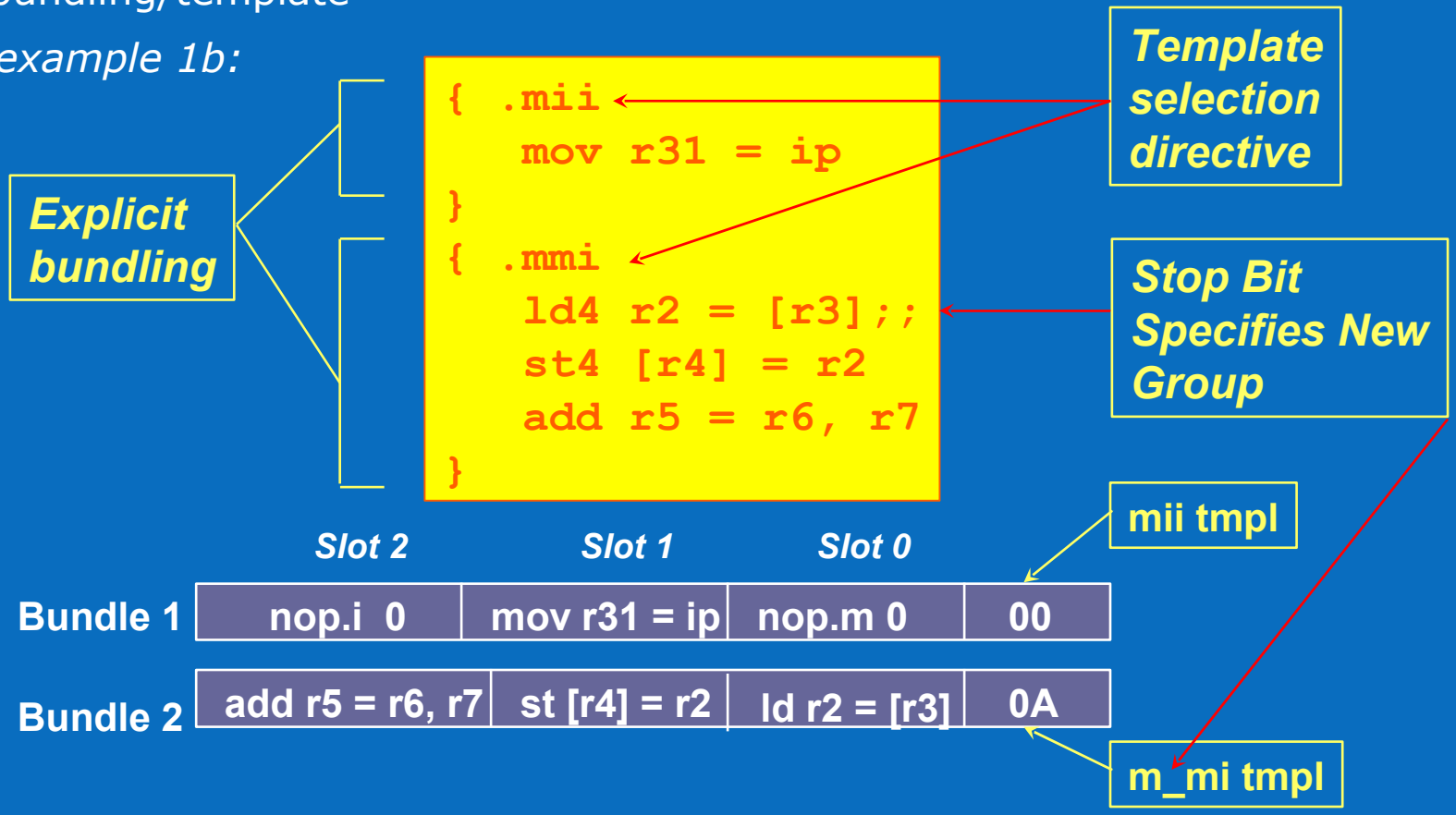
- Three instruction slots (41 bits each), a template field (5 bits)
- Instruction groups can span over several bundles
- *Example:*



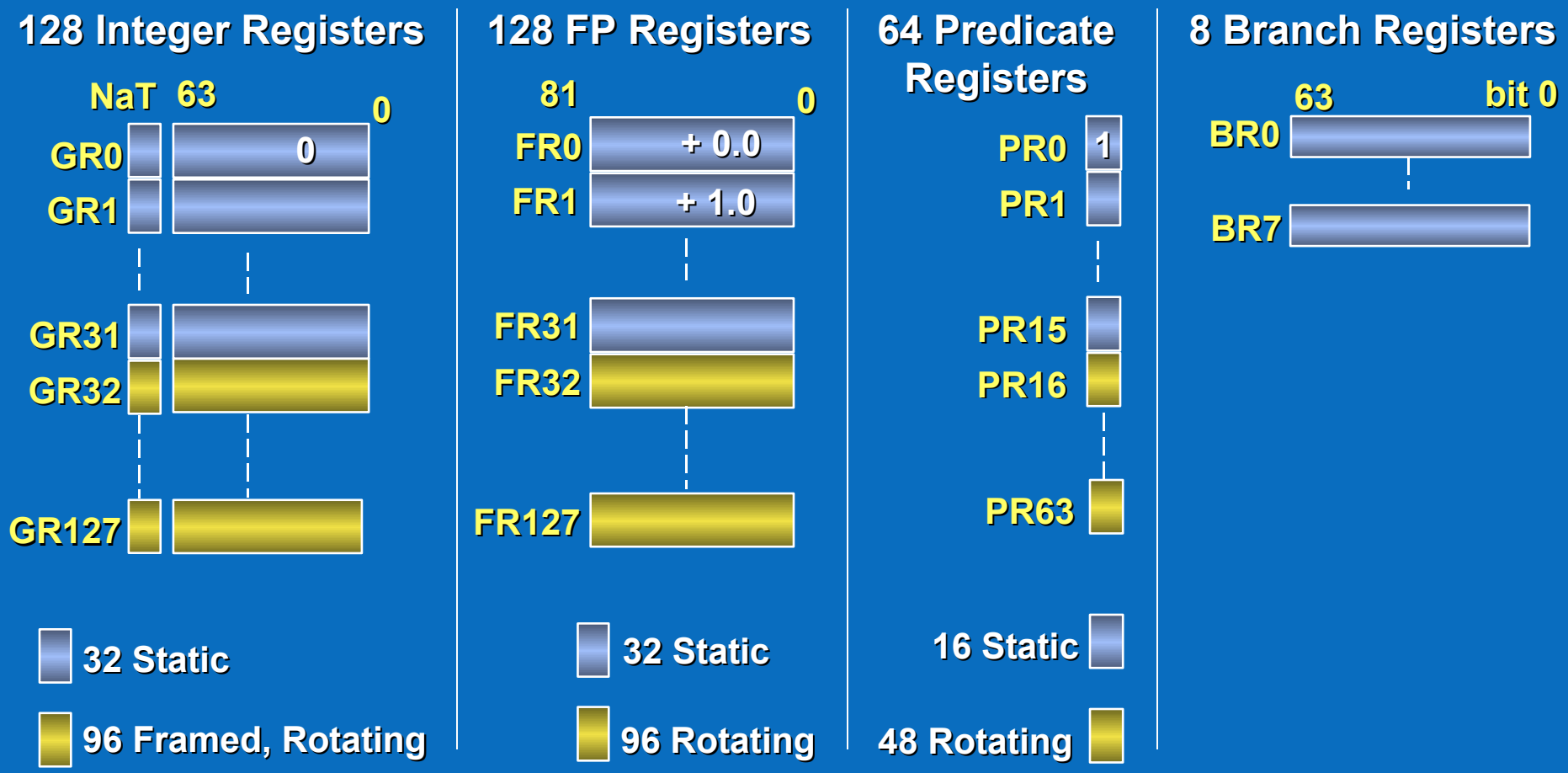
Instruction Level Parallelism

Explicit bundling/template

- *Code example 1b:*



Massive Register Set



Large number of registers enables flexibility and performance

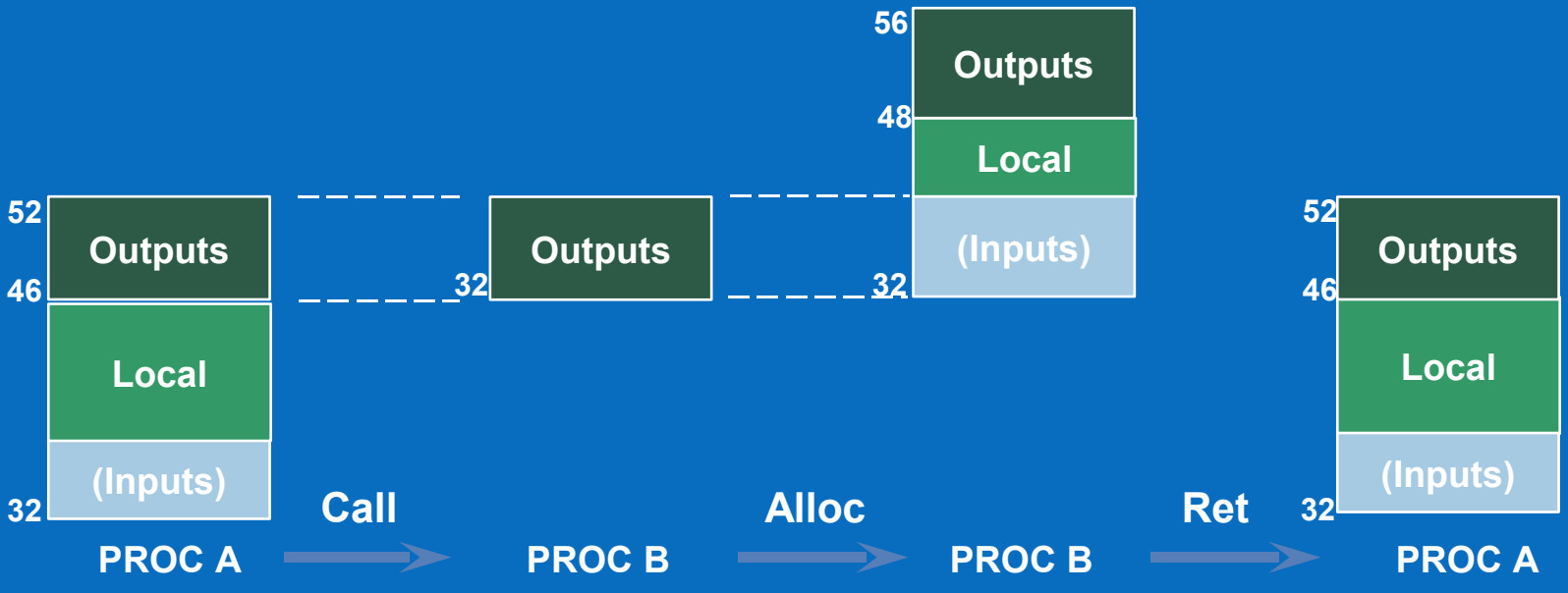
Register Stack

- **Call changes frame to contain only the caller's output**

Alloc sets the frame region to the desired size

Three architecture parameters: local, output, and rotating

- **Return restores the stack frame of the caller**



Avoids register spill/fill upon procedure call/return

Predication

Code Example: absolute difference of two numbers

C Code

```
if (r2 >= r3)
    r4 = r2 - r3;
else
    r4 = r3 - r2;
```

Non-Predicated Pseudo Code

```
        cmpGE   r2, r3
        jump_zero P2
P1:     sub  r4 = r2, r3
        jump end
P2:     sub  r4 = r3, r2
end:    ...
```

Predicated Assembly Code

```
     cmp.ge p1,p2 = r2,r3 ;;
(p1) sub r4 = r2,r3
(p2) sub r4 = r3, r2
```

Control Speculation

Code example: using control speculation

```
            cmp.lt  p1, p2 = r31, r2
(p2)        br.cond.spnt  exit
            ld8  r3 = [r4] ;;
            add  r6 = r3, r5
            ...
exit:       nop.b  0
```

Without speculated load and add

```
            ld8.s  r3 = [r4]  ;;
            ...
            add  r6 = r3, r5 ;;
            ...
            cmp.lt  p1, p2 = r31, r2
(p2)        br.cond.spnt  exit
            chk.s   r6, recv
next:       ...
exit:       nop.b  0

            // recovery code
recv:       ld8  r3 = [r4] ;;
            add  r6 = r3, r5
(p0)        br.cond.sptk  next
```

With speculated load and add

Use **chk.s** to test for deferred exception tokens.

Data Speculation

Code example: using data speculation

```
st4   [r2] = r31
ld4   r4 = [r3] ;;
add   r6 = r4,r5 ;;
sxt4 r7 = r6
```

Without advanced load

```
        ld4.a  r4 = [r3] ;;
        ...
        st4  [r2] = r31
        ld4.c.clr  r4 = [r3]
        add  r6 = r4,r5 ;;
        sxt4  r7 = r6
```

Advanced load with ld.c

```
        ld4.a  r4 = [r3] ;;
        ...
        add  r6 = r4,r5 ;;
        ...
        st4  [r2] = r31
        chk.a.clr  r4, recv
back:   sxt4  r7 = r6

recv:   ld4  r4 = [r3] ;;
        add  r6 = r4, r5
(p0)    br.cond.sptk back
```

Advanced load with chk.a

Use ***ld.c*** if only *load* is speculated, ***chk.a*** if *load* and its uses are speculated.

Floating-Point Architecture

Full IEEE support/Floating Point Register

- 82 Bit Floating Point Register
- Instruction Set Supports:
  - Single, double, double-extended data types

FMA - multiply-add instruction (f = a * b + c)

Example:

```
fma.d f42=f8,f36,f41
```

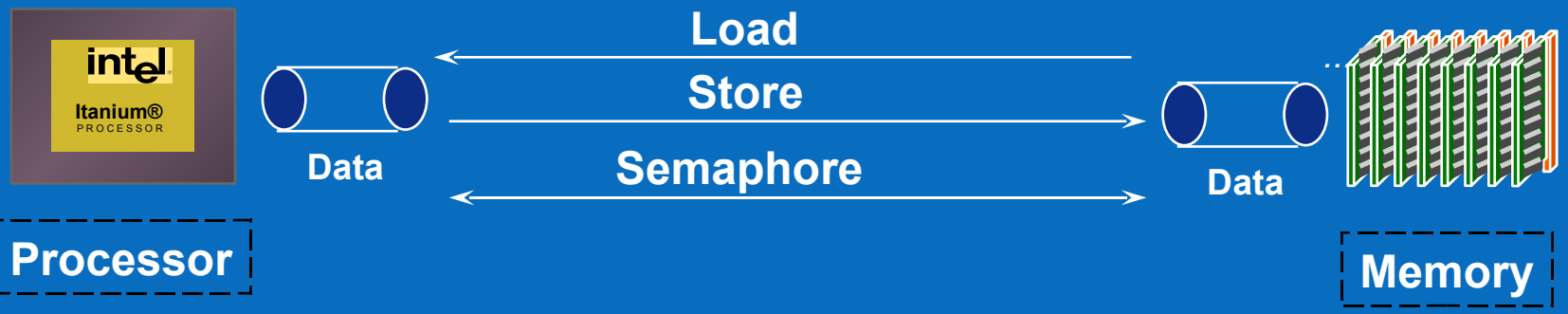
Arithmetic

- Support for software divide and square root using reciprocal and reciprocal square root approximation
- Max, min instructions for floating-point

Data transfer

- Load, store, GR <=> FR data transfer; load pair to double data

Memory Access Instructions



Memory addressing by byte (64-bit address)

- It is specified by the contents of a general register.

Byte ordering

- Big-endian or little-endian data accesses

Data elements should be stored (aligned) on natural boundaries

- Hardware fault is on misaligned references.
- 10-byte floats are stored on 16-byte boundary.

Architecturally Visible Memory Hierarchy

- Memory Instruction specify the cache level to occupy

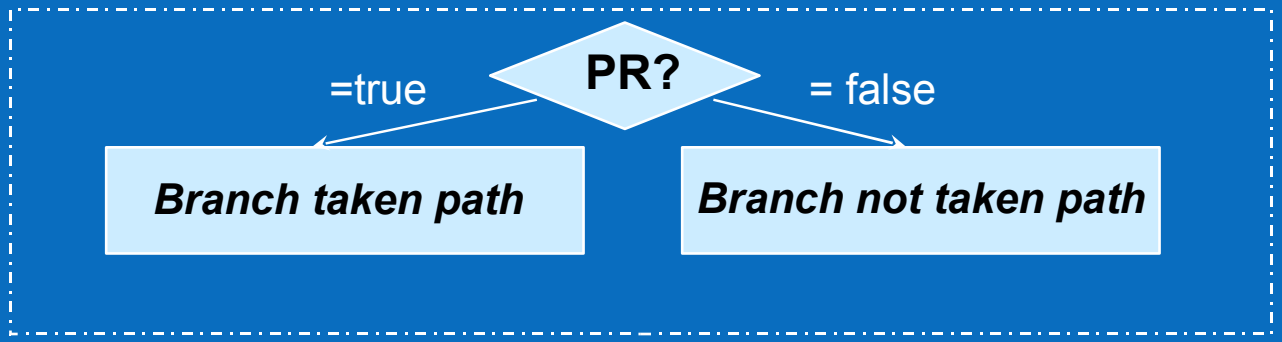
Software Prefetch Instructions

- Example: `lfecth.nts [GR23]`

Control Flow Instructions

Conditional branch

- Uses predicate to determine branching condition



- *Code example:*

C Code

```
count = 0;
do
   count = count + 1 ;
while (count < n);
```

Assembly Code

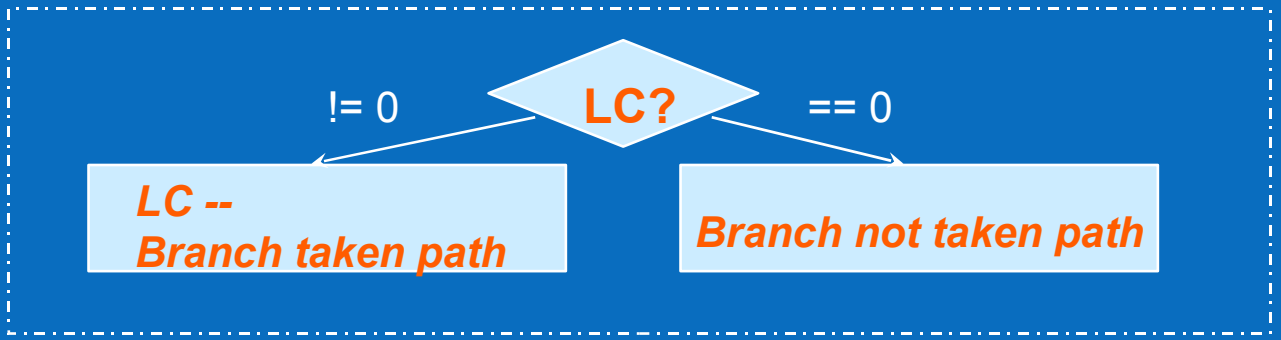
```
          mov  r31 = 0 ;;
loop:     add   r31 = 1, r31 ;;
          cmp.lt  p1, p2 = r31, r2
(p1)      br.cond.sptk  loop
```

Branch** hint-static **taken

Control Flow Instructions

Counted loops

- Use loop count (LC) application register



- *Code example:*

C code

```
for (i=0; i<10; i++) {
   ;
   }
```

Assembly code

```
        mov ar.lc = 9 ;;
loop:   nop.i  0
        br.cloop.sptk  loop
```

set LC = 9

if LC != 0, goto loop

Software Pipelining

- Consider

C code:
```
for (i = 0; i < n; i++)
        y[i] = a * x[i];
```

Pseudo Code:
```
loop:
        load xi
        fmul yi = a, xi
        store yi
        branch loop
```

- Assume
  - Instruction latencies:
    - load 4 cycles§
    - fmul 2 cycles§
    - store 1 cycle§
    - branch 1 cycle§
  - Load, fmul, store and branch can be issued in the same instruction group.

§Cycle counts for demonstration purposes only.

Software Pipelining

Cycle 1:	load x1		
Cycle 2:	load x2		
Cycle 3:	load x3		
Cycle 4:	load x4		
Cycle 5:	load x5	fmul y1=a,x1	
Cycle 6:	load x6	fmul y2=a,x2	
Cycle 7:	load x7	fmul y3=a,x3	store y1
Cycle 8:	load x8	fmul y4=a,x4	store y2
Cycle 9:		fmul y5=a,x5	store y3
Cycle 10:		fmul y6=a,x6	store y4
Cycle 11:		fmul y7=a,x7	store y5
Cycle 12:		fmul y8=a,x8	store y6
Cycle 13:			store y7
Cycle 14:			store y8

For n = 8

Prolog

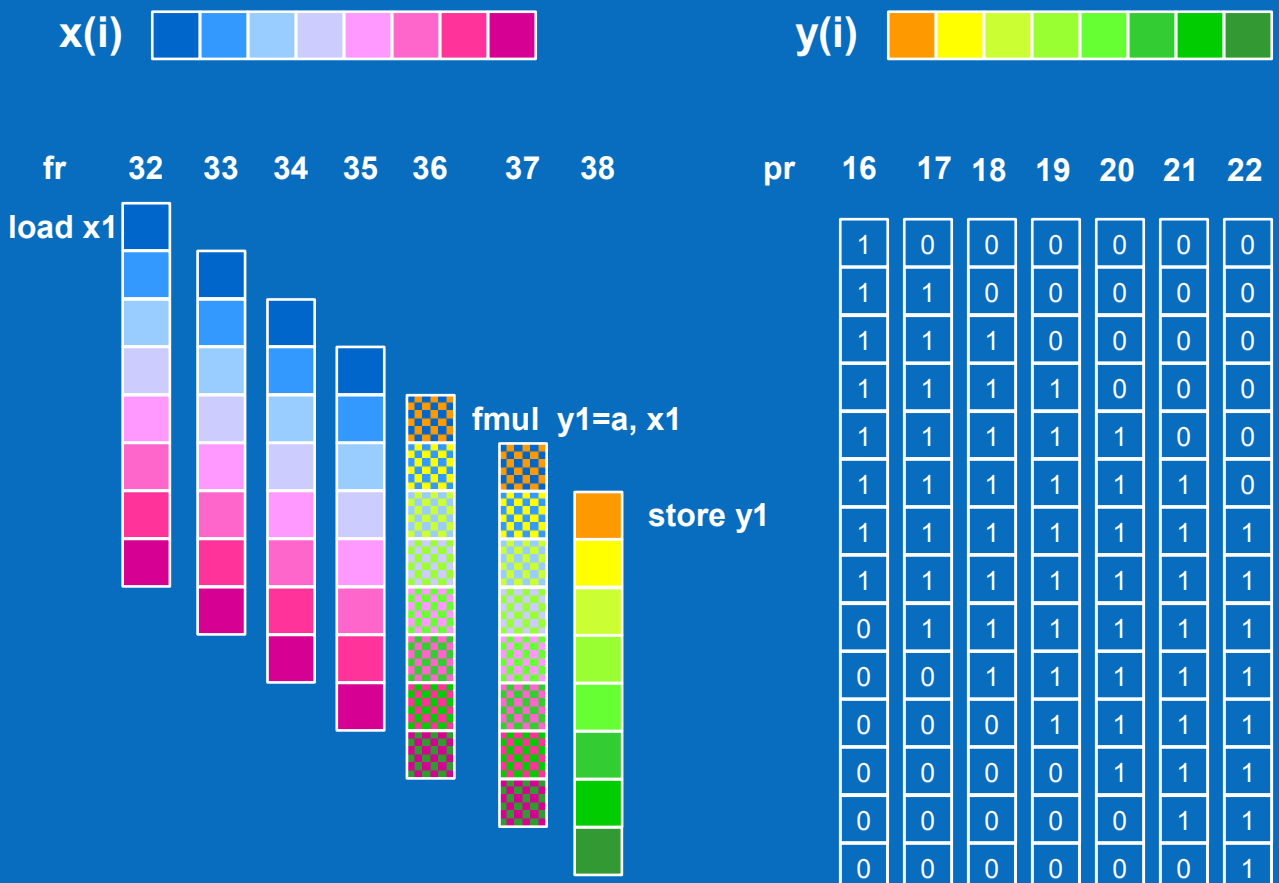
Kernel

Epilog

In this example, one iteration takes 7 cycles.

Cycle counts for demonstration purposes only.

Software Pipelining



Cycle counts for demonstration purposes only.

Software Pipelining

Actual code example:

```
    // Initialization
            mov pr.rot=0            // Clear all rotating pred regs
            cmp.eq p16,p0=r0,r0                 // Set p16=1
            mov ar.lc=7             // Set loop counter to n-1
            mov ar.ec=7             // Set epilog counter # of stages
            ...
loop:
            { .mfi
(p16)       ldfd f32=[r32],8                    // Stage 1: Load x
(p20)       fmpy.d f36=f6,f36                   // Stage 5: y=a*x
            nop.i 0
            }
            { .mfb
(p22)       stfd [r33]=f38,8                    // Stage 7: Store y
            nop.f 0
            br.ctop.sptk.few  loop              // Branch back
            }
```

Itanium®-Based System Environment

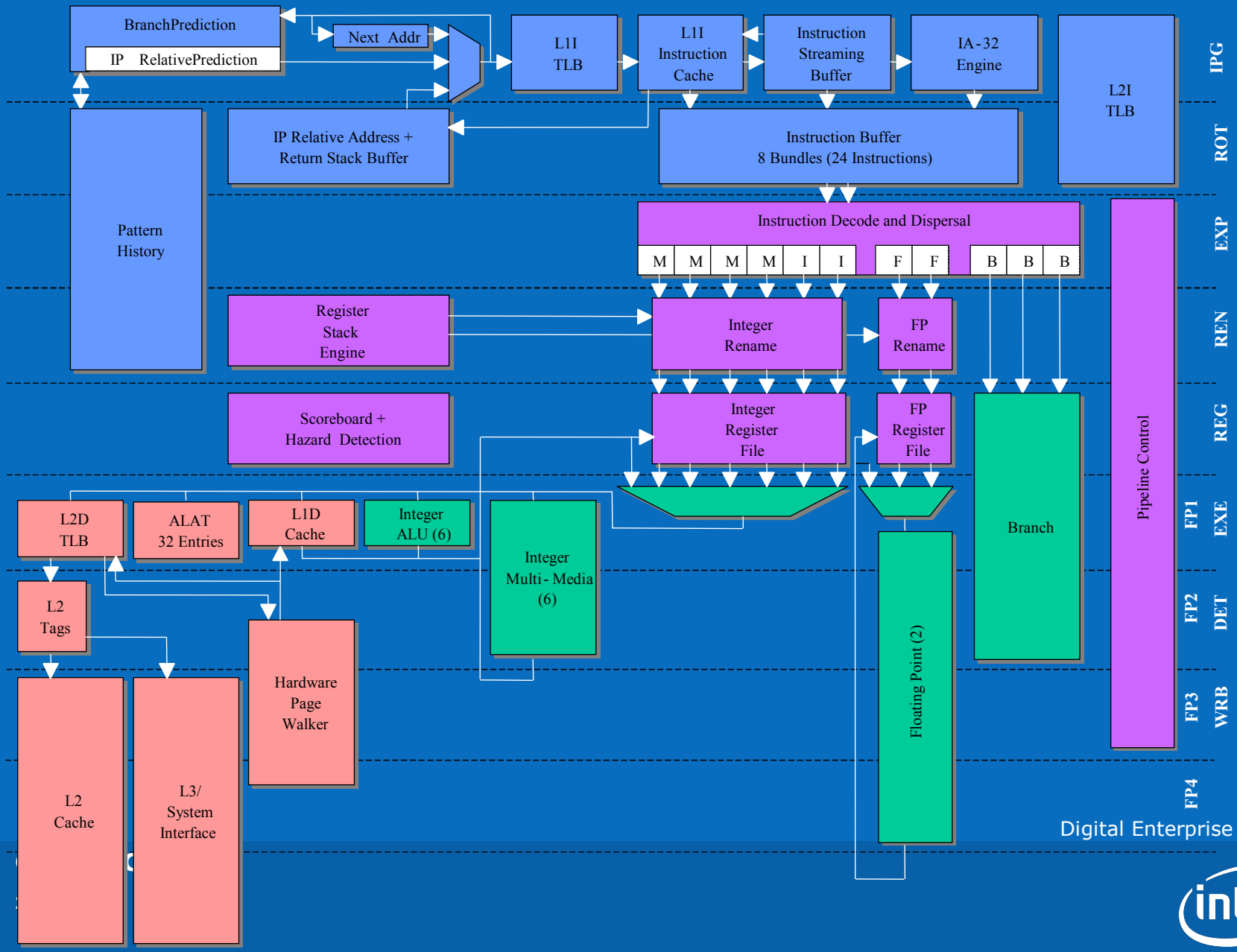
Itanium-based architecture provides support for operating systems to run both IA-32 and Itanium-based applications.

It supports Windows*, Unix*, VMS*, and Linux*.

Itanium-based applications take full advantage of the architectural features.

All Itanium-based operating systems support both x86 and EPIC instruction sets.

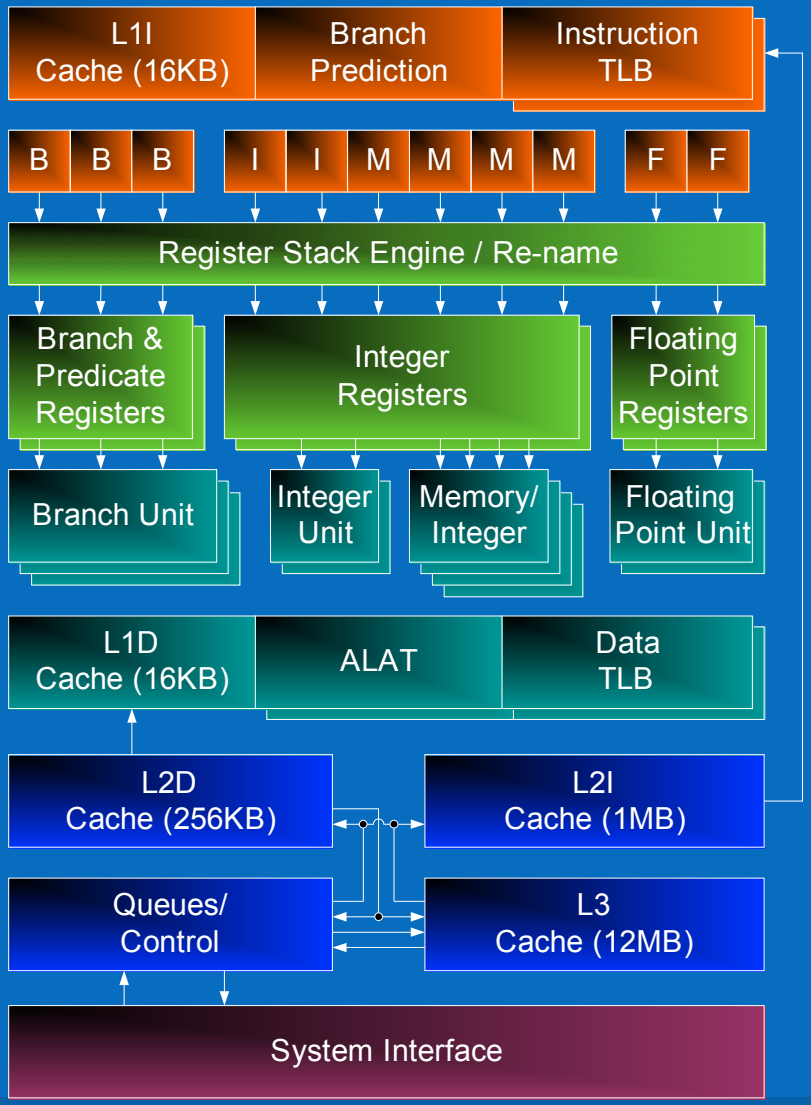
Itanium® 2 – Block Diagram and Pipeline



Digital Enterprise Group

intel

Montecito Micro Architecture



Summary

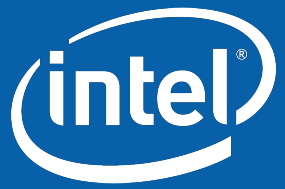
Intel® Itanium® Architecture Delivers Excellent Performance

Contains Many Next Generation Processor features

- i.e. Hardware Support for Software Pipelining

The Performance Features are Uncovered via The Software Development Tools

- Compilers (Intel® C++ and Fortran Compilers)
- Performance Analysis Tools (VTune™)



Back-up

Cache Structure

Very high speed memory for data that gets reused

Organized into “cache lines”

- Access of a single element brings in enough adjacent elements to fill the line (64/128 consecutive bytes)
- Underlying assumption that if you need one element you will need its neighbors soon

Cache lines are organized into “associative sets” or “ways”

- Greater associativity allows the hardware more flexibility in cache line replacement algorithms

Cache Structure

Very high speed memory for data that gets reused

Organized into “cache lines”

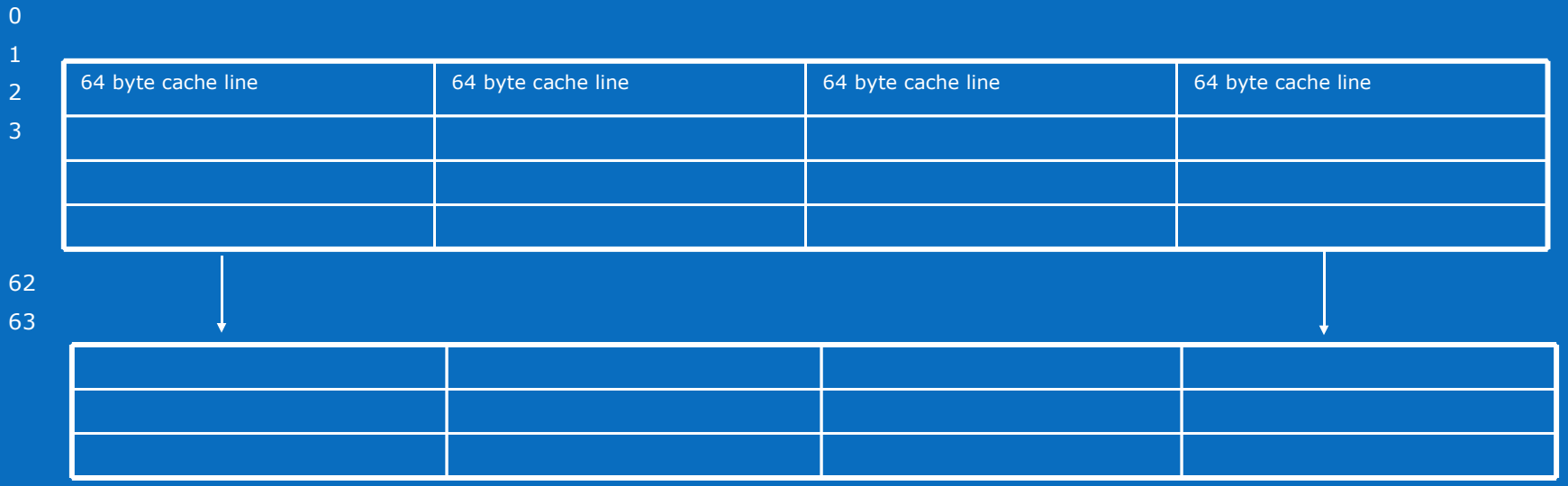
- Access of a single element brings in enough adjacent elements to fill the line (64/128 consecutive bytes)
- Underlying assumption that if you need one element you will need its neighbors soon

Cache lines are organized into “associative sets” or “ways”

- Greater associativity allows the hardware more flexibility in cache line replacement algorithms

L1D Cache Structure

Itanium® 2 Processor 16 KB 4-way associative L1 Data Cache (Integer Data)



L1D has one cycle latency for integer loads:

- **All cacheable integer loads go through L1D (write through)**
- **Use L1D micro pipeline to access general register file**
- **Cache line row determined by address bits 6 through 11**
- **64 associative sets**

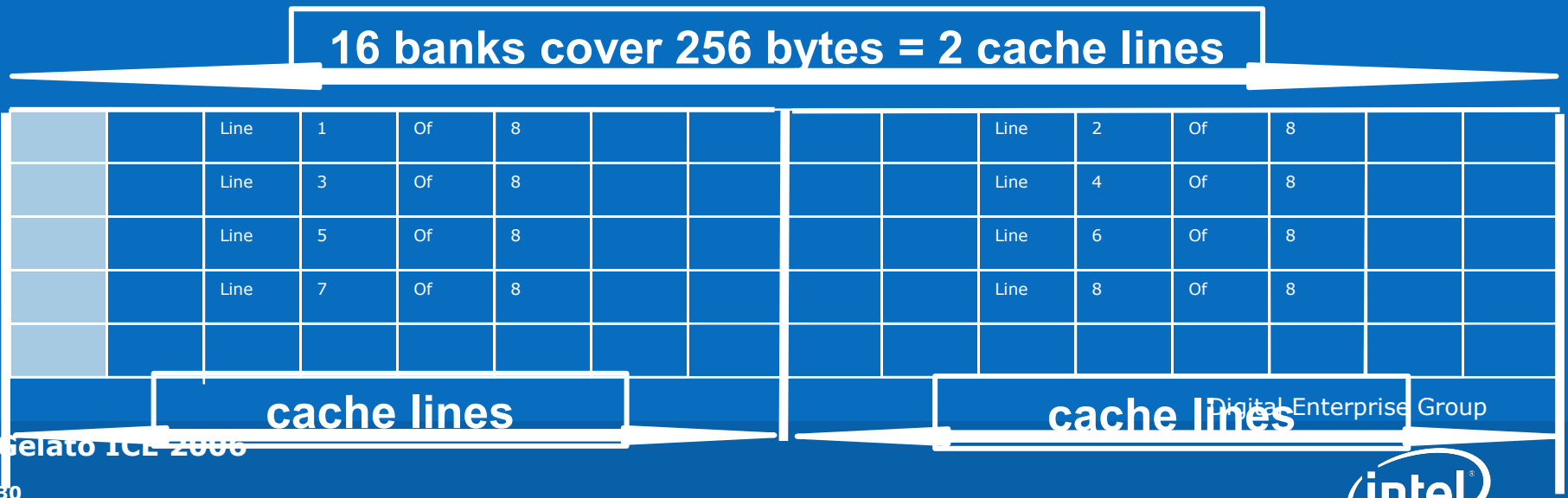
L2 Unified Cache Bank Structure

256KB, 128 byte cache lines, 8 way associativity

Each associative set is 1KB, 256 associative sets

Bank structure allows fast transfers from/to large Itanium® 2 Processor L2 Cache

16 banks each 16 bytes wide



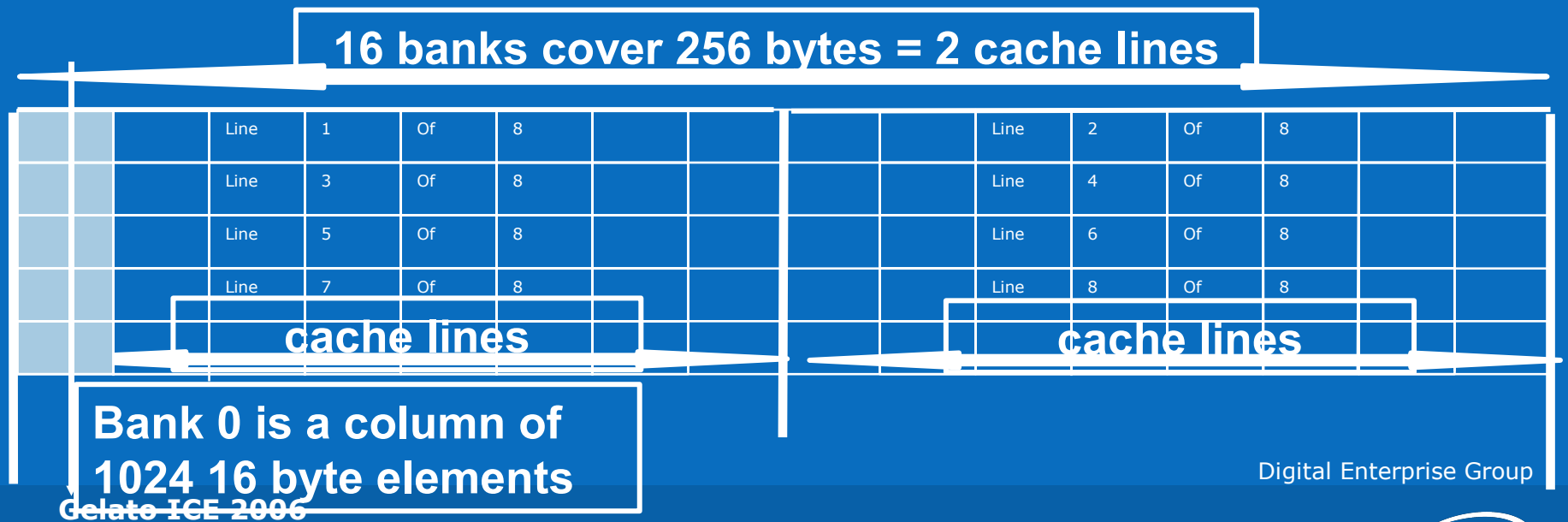
L2 Unified Cache Bank Structure

256KB, 128 byte cache lines, 8 way associativity

Each associative set is 1KB, 256 associative sets

Bank structure allows fast transfers from/to large Itanium® 2 Processor L2 Cache

16 banks each 16 bytes wide



Itanium® 2 Processor L2 Cache Access

L2 data access controlled by 32 entry queue (OzQ) and allows out of order data return

- FP data loaded to FP register file directly from L2

Minimum integer latency is 5 cycles

Minimum floating point latency is 6 cycles

Latency is increased by:

- Cache miss
- Bank conflicts cause OzQ cancels (measured to add 6 cycles)
- Multiple misses and misses to lines being updated will cause OzQ recirculates (measured to add ~17 cycles)
  - Only one data access is escalated to L3 and the system bus, the others recirculate

Virtual Memory on Itanium